Parallel evaluation of dataflow programs for extreme-scale many-task computing

Authors Hidden

Abstract

Developing programming solutions to help applications utilize the high concurrency of multi-petaflop computing systems is a challenge. Emerging languages such as Dryad, Swift, and Skywriting provide a promising direction. Their implicitly parallel dataflow semantics allow the high level logic of large-scale applications to be expressed in a manageable way while exposing massive parallelism through many-task programming. However, the implementations of these languages limit the evaluation of the overall program to a single-node computer, relying on threads and the multiple SMP cores of the evaluation node to generate the parallel tasks which are in turn executed on many nodes.

In this work, we provide a model for *distributed* parallel evaluation of dataflow programs in a manner that spreads the overhead of program evaluation and parallel task generation throughout an extreme-scale computing system. This execution model enables function and expression evaluation to take place on any node of a computing system. It breaks parallel loops into fragments for distributed execution, and can more readily achieve the task generation rates needed to efficiently utilize future exascale systems.

This paper describes the design and preliminary implementation of a distributed evaluation model for implicitly parallel dataflow programs, and motivates it with requirements projected from scientific applications. The preliminary implementation shows promising scalability curves and processes 500K task/sec on 1024 low-speed compute nodes.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords MPI, ADLB, Swift, Turbine, exascale, scripting, concurrency, dataflow, futures

1. Introduction

Exaflop computers capable of 10^{18} floating-point operations/s are expected to provide concurrency at the scale of O(10^9) threads on O(10^6) cores [22]. Such "extreme-scale" systems will enable and demand new problem solving methods that involve many concurrent and interacting tasks. Methodologies such as rational design, uncertainty quantification, parameter estimation, and inverse modeling all have this many-task property. All will frequently have aggregate computing needs that require exascale computers.

Running many-task applications efficiently, reliably, and easily on such extreme-scale computers is challenging. The many-task model may be split into two important processes: *task generation*, which evaluates a user program, often a dataflow script, and *task distribution*, which distributed the resulting tasks to workers. The user work is performed by *leaf functions*, which may be implemented in native code or as external applications. This computing model draws on recent trends that emphasize the identification of coarse-grained parallelism as a first and separate step in application development [12, 24, 25]. Leaf functions themselves may be multi-core or even multi-node tasks.

Currently, many-task applications are programmed in two ways. In one, the logic associated with the different tasks is integrated into a single program, and the tasks communicate through MPI messaging (where they exist in different memory spaces) or function calls (as in the parallel version of the Common Component Architecture, CCA [3], where components exist in the same memory space.) This approach uses familiar technologies but can be inefficient unless much effort is spent on incorporating load-balancing algorithms into the application. Moreover, the approach can involve considerable programming effort if multiple component codes have to be tightly integrated. Load balancing libraries based on MPI, such as the Asynchronous Dynamic Load Balancing Library, ADLB [11] or on Global Arrays, such as Scioto [7] have recently emerged as promising solutions to aid in this approach. They provide a master/worker system with a put/get API for task descriptions, where workers may dynamically add work to the system. However, they lack a comprehensive programming model, data model, and other features required for high productivity programming.

The second approach is that a script or workflow is written that invokes the tasks, in sequence or in parallel, with each task reading and writing files from a shared file system. Examples include Dryad, Skywriting, and Swift. This approach is convenient for the user, particularly when each task is a distinct executable program. However, performance can be poor, as existing many-task scripting languages are implemented with *centralized evaluators* that are not capable of sustaining the high overall task rate necessary to efficiently utilize O(10^6) cores.

Consider an example application in Swift:

```
Model m[];
Statistics s[];
foreach i in [0:999999] {
  s[i] = runModel(m[i]);
}
```

This parallel *foreach* loop runs 10^6 independent instances of the model. In a dataflow language like Swift, this foreach loop generates 10^6 parallel tasks. However, the *evaluation* of the foreach loop itself takes place on a single compute node (typically a cluster

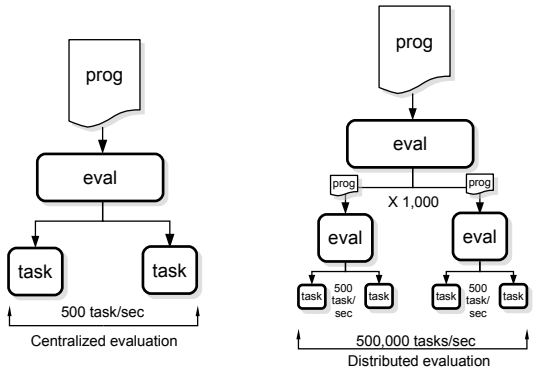

Figure 1. Distributed parallel evaluation (right) increases scalability over centralized evaluation (left).

"login node"). Such nodes, even with many cores (today ranging from 8 to 24) are only able to generate about 500 task invocations per second. Task distribution systems such as Falkon [18] achieve 3,000 tasks/second with more parallel resources. And despite the fact that the Swift language exposes abundant task parallelism, when the evaluation of the language is constrained to take place on a single compute node, task generation rates can be a significant scalability bottleneck. An application that needs to run 10^{12} 10-second tasks across 10^6 cores would need to run 1000 tasks per core, and initiate 10^8 tasks per second to keep that many cores fully utilized. *This is many orders of magnitude greater than the rates achievable with single-node dataflow language evaluation.* If 10^5 processes, 0.01% of the system, are allocated as control processes, each control process is responsible for generating 1000 tasks per second.

This paper describes an approach capable of generating and distributing tasks at this scale. Its innovation is based on expressing the semantics of parallel dataflow in a language-independent representation with semantics similar to the Swift parallel scripting language, and implementing a distributed evaluation engine for that representation which decentralizes the overhead of task generation. The engine is called "Turbine" and its intermediate code input is called "TIC". Turbine execution employs distributed dependency processing, task distribution via ADLB, and a distributed in-memory data store that makes TIC objects accessible from any node of a distributed memory system. The approach combines the performance benefits of explicitly parallel asynchronous load balancing with the programming productivity benefits of implicitly parallel dataflow scripting.

The remainder of this paper is organized as follows. In §2, we motivate this work by providing two representative examples of scripted applications. In §4, we discuss existing and prior work related to these problems. In §5, we describe our design for distributed parallel evaluation of TIC. In §6.2 and §6.3, we present the Turbine layer in detail. In §7, we report performance results from the use of the implementation in various modes. In §8 and §9, we discuss the status of our work, and offer concluding remarks.

2. Motivation: Many-task Applications

Parallel scripted representation of applications for many-task computations is convenient from the perspective of both the user and the underlying evaluation engine. The user is presented with a rapidly prototyped, compact yet complete application flow. Traditionally, scripts have been imperative programs, with some exceptions such as `make`. Under futures variable semantics, the evaluation engine is presented with many, asynchronous dataflow dependent tasks to be distributed to the execution environment. This section presents many-task applications that have been conveniently represented by parallel scripting paradigm and have produced parallel tasks in the order of hundreds of thousands.

2.1 Application: Molecular Analysis in Biochemistry

The diagram in Figure 2 shows a schematic of *modftdock*, a protein docking application. This is a three-stage workflow involving an irregular granularity and computational load. It performs *N* dock tasks on each of *M* molecules, then for each molecule, merges the dock results and scores them. Table 1 provides performance information on the tasks. The tasks listed for each stage can all be performed concurrently provided the data dependencies at each task are met. A slow task distribution rate can cause a bottleneck in such a case where an intermediate task (*modmerge*) is of short duration, but its completion is required in order to begin the following significantly-longer task (*score*). Unless such intermediate tasks are dispatched at a high rate, the available compute cores can not be optimally utilized, and execution of the overall application will be delayed. Table 1 shows one such projected scenario where an efficient task dispatch rate would keep 10^6 cores busy.

2.2 Application: Branch-and-bound Optimization

Power-grid distribution design is an example of a problem that involves solving a single integer non-convex optimization problem. The initial solution reveals how to subdivide the domain (*branching*), then a new tighter approximation is constructed on the subdomain and solved. The process is repeated. The solution of the problem on a given subdomain is guaranteed to not exist (bounding) unless it corresponds to a globally optimal solution. Pseudocode to solve a branch-and-bound problem is presented in Listing 1. The parallel branch-and-bound method has a dynamic computational profile: for problems with partial differential equation (PDE) constraints, 1,000 subdomains could be solved, each using 1,000 processors, allowing an effective use of 1M processors simultaneously.

```
Queue q; Problem p; Solution s;
double upperBound = Infinity;
q.enqueue(0);
while (q.top) {
  p = q.dequeue;
  s = optimize(p);
  if (s.solution != Infeasible) {
    if (s.solution == Integer) {
      if (s.objective < upperBound) {
        upperBound = s.objective; //bound
      }}
    else {              //branch
      q.enqueue(s.left);
      q.enqueue(s.right);
    }
  }
}
```

Listing 1. Pseudo code for the branch-and-bound Optimization

2.3 Application Requirements

In addition to the above mentioned applications, ensemble studies involving different methodologies such as uncertainty quantification, parameter estimation, massive graph pruning and inverse modeling all have a requirement of being able to generate and dispatch tasks in the order of millions to the distributed resources. We have been investigating some of these classes of applications, such as the Decision Support System for Agrotechnology Transfer (DSSAT) aimed at land-usage modeling and Soil and Water Assessment Tool (SWAT) based studies in the field of Hydrology. On the

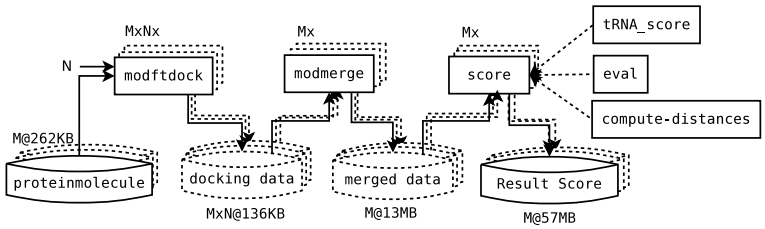

Figure 2. The modftdock application flow; M and N denotes the number of proteins and the number of iterations per protein respectively

Application	Stage	Measured		Required	
		Tasks	Task Duration	Tasks	Task Emission Rate
modftdock	dock	1,200,000	1,000s	10^9	10^{12}/s
	modmerge	12,000	5s	10^7	2×10^{12}/s
	score	12,000	6,000s	10^7	1.33×10^{12}/s
Power-grid Distribution	economic-dispatch	10,000	15s	10^8	6×10^{12}/s
DSSAT	runDSSAT	100,000	12s	10^8	8×10^{12}/s

Table 1. Quantitative description of applications and required performance on 10^6 cores

basis of the computational properties these application classes display, we believe, performance and resource utilization will increase significantly when these applications are ported using Turbine.

3. Building Blocks: Swift and ADLB

Two technologies motivate and provide a foundation for the work described here. Swift provides a productive programming model for the high-level code of many applications, and the Asynchronous Dynamic Load Balancing (ADLB) library provides the means to take the Swift programming model to exascale performance levels.

Swift [4, 26] is a widely deployed parallel scripting language for scientific computing. The Swift language is typed and concurrent, providing multiple features to support distributed scientific batch computing, include data structures (arrays, structs), string processing, use of external programs, external data access to filesystem structures, etc. Swift programs are compiled into the Karajan workflow language and interpreted by a runtime system based on the Java CoG Kit [23]. Other work investigated additional languages based on the Swift runtime system, such as Python [2] and R [19, 21]. While Swift is capable of generating and scheduling thousands of tasks, and managing their execution on a wide range of distributed resources, each Swift script is evaluated on a single node, resulting in a performance bottleneck. Removing such bottlenecks is the primary motivation for the work described in this paper.

The Asynchronous Dynamic Load Balancing (ADLB) [11] library is an MPI-based library for managing a distributed work pool at large scale. ADLB applications use a simple put/get interface to deposit "work packages" into a distributed work pool and retrieve them. Work package types, priorities, and "targets" allow this interface to implement sophisticated variations on the classical master/slave parallel programming model. ADLB is efficient (up to 25,000 work packets per second per node on an Ethernet-connected Linux cluster) and scalable (the nuclear physics applications described in [11] has run on 131,000 cores on an IBM BG/P). ADLB is used as the load balancing component of Turbine, where its work packages are the indecomposable tasks generated by the system. The implementation of its API is invisible to the application (Turbine, in this case) and the work packages it manages are opaque to ADLB. An experimental addition to ADLB has been developed to support Turbine's shared variables, implementing a publish/subscribe interface.

4. Related Work

Scioto [7] is a lightweight framework for providing task management on distributed memory machines under one-sided and global-view parallel programming models.

Skywriting [13] is a coordination language that can distribute computations expressed as iterative and recursive functions. It is a dynamically typed language that offers limited data mapping mechanisms through a static file referencing; our language model offers a wider range of static types with a rich variable-data association through its dynamic mapping mechanism. The underlying execution engine, called CIEL [14] is based on a master-worker computation paradigm where workers can spawn new tasks and report back to the master. As of this writing, Skywriting/CIEL lacks support for Message-Passing based computation, which limits its deployability on a large class of distributed systems. Thanks to ADLB, our framework integrates Message-Passing support in the core execution engine. While scaling benchmarking results of the Skywriting/CIEL framework are shown on wide-scale distributed systems (e.g., clouds), no distributed interpretation/evaluation strategy is known to exist for task generations through Skywriting scripts.

MapReduce [6] also provides a programming model and a runtime system to support the processing of large-scale datasets, Sawzall [17] is an interpreted language that builds on MapReduce and separates the filtering and aggregation phases for more concise program specification and better parallelization. MapReduce/-Sawzall share our goal of providing a programming tool for the specification and execution of large parallel computations on large quantities of data and facilitating the utilization of large distributed resources, but the MapReduce programming model just supports key-value pairs as input or output datasets and offers two types of computation functions, map and reduce; we use a type system and allow the definition of complex data structures and arbitrary computational procedures. Dryad [9] is an infrastructure for running data-parallel programs on a parallel or distributed system. In addition to allowing files to be used for passing data between tasks, it allows TCP pipes and shared-memory FIFOs to be used. Dryad graphs are explicitly developed by the programmer; our graphs are

implicit, and the programmer doesn't have to worry about them. Furthermore, Dryad does not allow iterative computation which are easily expressed by Swift with looping constructs. A scripting language called Nebula was originally developed above Dryad, but it doesn't seem to be in current use. Scripting-level use of Dryad is now supported primarily by DryadLINQ [27], which generates Dryad computations from the LINQ extensions to C#.

Spark [28] is an enhanced implementation of the Map-Reduce paradigm in that it adds the support for iterative computations. A distributed fault-tolerance layer called the Resilient Distributed Dataset (RDD) is built into the framework that compensates for failed nodes while maintaining the state of the computation.

Linda [1] introduced the idea of a distributed tuple space, which is a key concept in ADLB and Turbine. This idea was further developed for distributed computing in Comet [10], which has a similar goal as our work, but focuses on distributed computing and is not concerned with scalability on HPC systems. Lithe [16] is a low-level framework that supports the development of parallelizing libraries by taking advantage of the underlying Operating System's exposed resources.

The development of scripts or code to link together executable tasks has been the subject of a fair amount of previous effort. Examples of this are PCN [8], CORBA [15], and CCA [3]. These systems generally assume large blocks of code need to be infrequently linked, and are not concerned with the performance implications of large numbers of executions.

Other related projects include: the Standard Template Adaptive Parallel Library (STAPL) [20], a parallel programming framework that extends C++ with support for parallelism; DAGuE [5], a framework for scheduling and management of tasks on distributed and many-core computing environments; and Lithe [16], a low-level framework that supports the development of parallelizing libraries using the underlying operating system's exposed resources.

In conclusion, there are a significant number of parallel task distribution frameworks and languages. These frameworks provide/-support/facilitate mechanisms/techniques to generate, dispatch and execute dataflow coherent tasks over a distributed computing infrastructure. In general, these frameworks lack a suitable mechanism for rapid task distribution, unlike our approach.

5. Distributed Parallel Dataflow Evaluation

We describe here the distributed parallel evaluation model that Turbine uses to execute programs with implicitly parallel dataflow semantics. Turbine's function is to interpret an intermediate code representation (Turbine Intermediate Code, or TIC) of a parallel dataflow program. TIC specifies semantics which are similar to the Swift parallel scripting language. We believe that TIC is general enough to serve as a model for performing the distributed parallel evaluation of many similar languages.

The main aspects of TIC are:

1. **Implicit, pervasive parallelism:** Most TIC statements are relations between single assignment variables. Variables may be open (unset) or closed (set to a value). Dynamic data dependency management enables expressions to be evaluated and statements executed when their data dependencies are met. Thus all functions in a TIC program are conceptually executable in parallel, limited in practice by throttling and by available resources.
2. **Typed variables and objects:** TIC variables and objects are constructed with a simple type model comprised of the typical primitive scalar types, a container type for implementing arrays and structures, and a scalar type for mapping external files and objects to in-memory variables.
3. **References to external data:** Variables may be "mapped" to external data in files, as in Swift and PyDFLow. We don't discuss here the mechanism by which mapping takes place, but assume that mappings are stored as an attribute of file-type variables.
4. **Constructs to support external execution:** Most of the application-level work in TIC is performed by external user application components (programs or functions); TIC is primarily a means to specify the parallel composition of the application components.

5.1 The Swift Evaluation Model

To underscore the motivation for this work, it is useful to briefly summarize the Swift program evaluation model [26]. A Swift program starts in a global scope. All statements in the scope are allowed to make progress concurrently through the application of a lightweight-thread event engine. Statements that block on unset variables are recorded and entered into a map of links from input variables to a suspended stack frame. When the inputs are ready, the suspended stack frame is restarted. Function invocations cause the creation of new scopes (stack frames) in which new local variables are dynamically created. When variables are returned by functions, they are closed. Input and output variables passed to functions exist in the original caller's scope, and are passed and accessed by reference. Since input variables cannot be modified in the called function, they behave as if passed by value.

The Swift evaluation model is capable of supporting large parallel scripts that run on thousands of processor cores. However, the evaluation (i.e., interpretation) of the script itself is constrained to run on a single node. This limitation is in large part due to the fact that Swift's future map is a shared in-memory data structure that resides in a single node, and that no mechanism exists for cross-address-space communication and synchronization of the state of future objects. The execution model of Turbine eliminates the limitations of this centralized evaluation model.

5.2 Overview of TIC

TIC operations include data manipulation, the definition of data-dependent execution expressed as *rules*, and higher-level constructs such as loops and function calls. TIC programs consist of *variables*, *statements*, and *functions*. Variables are typed in-memory representations of user data that may be mapped to files. Variable values may be scalars such as integers or strings or containers representing language-level features such as arrays or structures (records). All TIC variables are single-assignment futures: they are open when defined and closed after a value has been assigned. Containers are also futures, as are each of their members. Containers can be open or closed; while they are open, their members can be assigned values. Containers are closed by an explicit close operation. In practice is generated in the translation from the source program by scope analysis and the detection of the last write, the end of the scope in which they are defined, as any returned containers must be closed output variables. *Scopes* (i.e., stack frames) provide a context for variables for use as function and loop body activation records. Scopes are composed into a linked stack of "activation records."

Statements in TIC cause execution to occur and values to be assigned to variables. Statement types are assignment, conditional execution, and parallel iteration over collections. Statements serve as links from task outputs to inputs, forming an implicit task graph as the script executes.

TIC statements which require variables to be set before they can execute (primarily the primitives which call an external user application function or program) are expressed as the target operations of rules which specify the action, the input dependencies required

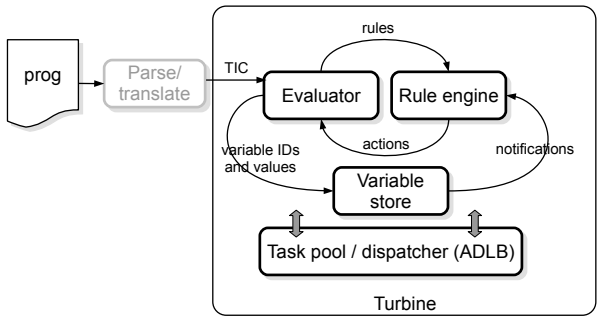

Figure 3. Turbine component architecture.

for the action to execute, and what output data objects are then set by the action. Building on our earlier example in Swift:

```
Model m = readModel();
Statistics s1 = runModel(m, 1);
Statistics s2 = runModel(m, 2);
Plot p = plotModel(s1, s2);
```

In this case, the model is run with two different random seeds, 1 and 2. These statements may be executed concurrently as soon as m is closed. The resulting TIC statements are shown below:

```
when { } then {m} from {readModel m }
when {m} then {s1} from {runModel m 1}
when {m} then {s2} from {runModel m 2}
when {s1 s2} then {p} from {plotModel s1 s2}
```

The `when` statements correspond to rule definitions stored in the distributed Turbine system. Data definitions, not shown, correspond to addresses in the global data store. When m is set by the worker processing the first action, rule engines responsible for dependent actions are notified, and progress is made.

The high-level architecture of Turbine is shown in 3. One of Turbine's main innovations, which facilitates distributed evaluation, is a distributed variable store based on futures. This store, accessible form any computing node within a TIC program, enables values produced by a function executing on one node to be passed to and consumed by a function executing on another, and manages the requisite event subscriptions and notifications. Rules are implemented a *rule engine* which in turn uses the subscription and notification services of the variable store. All three components in this architecture are implemented as multiple distributed parallel processes (in an MPI sense), as described in §6.

As above, the user starts with a source program (e.g. in Swift). This is translated into Turbine intermediate code, which is largely a straightforward parsing and flattening operation with a small amount of semantic analysis (e.g. to determine when collections can be closed).

All TIC execution takes place within TIC functions, and proceeds as follows.

- A procedure executes by sending each action that requires rule interpretation to the rule engine, and each action that involved a real or generated procedure call to an evaluation engine, and performs immediately a small number of primitive actions.
- These actions require variables to be closed:
  - Calling a function requires input vars to be closed.
  - Evaluating a conditional expression (the result of compiling an if() or switch() statement) requires the variable that represents the test condition to be closed.
  - Completion (but not initiation) of a segment of a foreach loop requires all of its members to be closed.
  - Completion of a foreach loop requires all of its segments to be completed.
  - Completion of a function call requires that all of its sub-calls are completed and their resources freed.
- When a function is invoked, a stack frame is created for its local variables. Input parameters and output results are held in a stack frame higher up in the calling chain. Input parameters cannot be modified. The outermost function is main() as in C, but has no arguments (external arguments are passed as in Swift using an access function `arg()`).
- Execution of the parallel `foreach` iterator is performed by creating a special rule to wait for a (dynamically determined) portion of the members of the associated collection to be set.
- The execution of a `foreach fragment` is performed as if the body of the loop is a function which has access to any variables it references in the parent scope as if they were passed as parameters.
- Any action which can be executed immediately, such as stack frame creation and variable initialization, is performed on entry to the function. Then all concurrent work activities such as evaluation of rules are sent back into the task distributor, to be routed to an available evaluation process.
- Any actions which depend on data such as conditional calls, expression evaluation (including calls to built-in primitives) and evaluation of application functions (app calls) are expressed in TIC as rules and sent to the rule engine to wait on their data dependencies. When these are met, the rule engine sends the action as a task to an evaluation engine.
- When all the actions initiated by a function (i.e., at or below the current stack frame) have completed, any collections created on the stack frame are freed, the stack frame and all its scalar variables are released, and the function is considered completed.

An example of this process is shown in Figure 4. First, the user script is translated into the Turbine intermediate code (TIC) format ①. In this example, a `foreach` rule is evaluated, resulting in the interpretation of a distributed loop operation, which is evaluated, producing additional execution units containing script fragments. These fragments are distributed using the task distribution system ② and are evaluated by rule engines elsewhere, resulting in the evaluation of application programs or functions, which are leaf functions of the TIC task graph ③. These obtain data from the data store and perform local work, for example, by calling into a native code routine ④.

6. Implementation of Turbine

We describe here the structure of our current first prototype implementation of Turbine, which is close in nature to the more abstract (and complete) design described in the prior section.

6.1 Program Structure

In our prototyping work, we leverage Tcl as the source language, and express TIC operations as Tcl commands ("procs"). As our focus is on refining the design of an intermediate representation for the optimal distributed operation of Swift-like languages, we have not yet adapted the Swift parser to produce TIC, and manu-

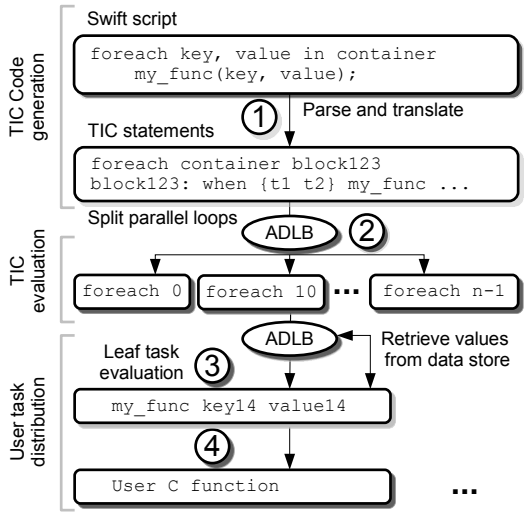

Figure 4. TIC execution in Turbine.

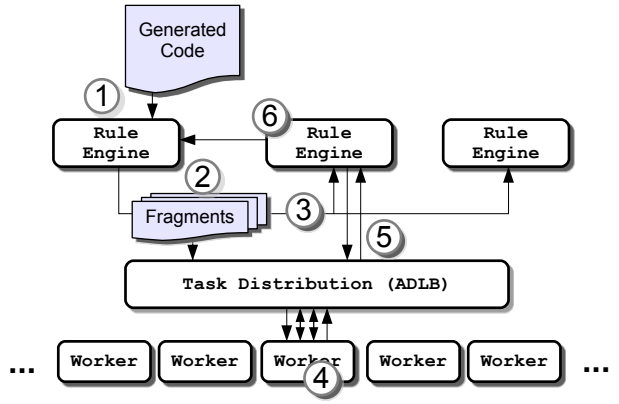

Figure 5. Distributed components in Turbine futures mechanics.

ally translate sample Swift test programs into TIC. We created a Tcl interface to ADLB, and thus have a convenient way to create distributed memory SPMD representations of TIC. These are standard MPI programs that can be launched by `mpiexec`. The programs we generate thus rely on two supporting libraries: a task creation API (Turbine proper) and a task distribution API (ADLB).

The task creation system evaluates the user script to produce execution units for distribution by the distribution system. Thus in the active process, new parallel work is generated on entry to most function calls, and on encountering loop fragments within a function call that can be efficiently split further. Turbine uses the ADLB API to create new tasks and to thus distribute the computational work involved in evaluating parallel loops and function calls - the two main parallelizable language constructs.

ADLB performs highly scalable task distribution, but does incur some amount of client overhead and latency to ingest tasks. As more ADLB client processes are employed to feed ADLB servers, this overhead becomes more distributed and the overall task ingestion and execution capacity increases. Each client is attached to one server, and traffic on one server does not congest other server processes. ADLB can scale task ingestion fairly linearly with the number of servers on systems as large as a 131K core BG/P. Thus a primary design motivation of Turbine is to optimize the number of TIC clients to maximize the ADLB task processing rate and thus the utilization of extreme-scale computing systems.

6.2 Rules and Distributed Futures

The fundamental system mechanics required to perform the operations required by the previous section were developed in Turbine, a novel *distributed future store*. The Turbine implementation comprises a globally-addressable data store, a rule evaluation engine, a subscription mechanism, and an external application function evaluator. It is implemented as an MPI program using the ADLB API, with each MPI process (rank) acting in one of the following roles.

There are two kinds of Turbine processes, each of which makes calls into ADLB to both receive and issue work tasks:

- **Turbine rule engines:** These processes are responsible for maintaining the rules that specify all dataflow dependencies. When the input conditions of a rule are met, the rule engine posts that rule's action as a Turbine work unit to ADLB.
- **Turbine action evaluators:** These processes receive rule actions from ADLB and perform them. Some actions result in the submission of new rules; other actions are leaf functions.

The ADLB library is responsible for receiving and distributing execution units throughout the system. As noted above, the ADLB server has been extended (experimentally) in this work to support Turbine data store operations as well as its normal services. Turbine data items contain an identifier, a type, a status, and a value. Unique data identifiers may be obtained from servers. When created, the data item has the identifier, type, and status "open". The location of the data item is determined through a simple hash scheme that places the item on one of the servers. The type of the item may be string, integer, float, or one of the complex types (file, container) described in §6.3. Data may be read or written by any component.

Each rule engine maintains a list of *rules*, each of which is contains an identifier, input list, output list, and specification of what to do when the inputs are ready. Each task executed by the system is associated with exactly one rule, thus, the rule is an important part of the implementation of the futures mechanism. Rule engines subscribe to required data items on servers as necessary. When ready, work is posted to the system through the server. Such execution units may be external application functions for workers or script fragments for evaluation by other rule engines.

Progress is made when futures are set and rule engines receive notification that new input data is available. Turbine uses a simple subscription mechanism whereby rule engines notify servers that they must be notified when a data item is ready. As a result, the engine either finds that the data item is already closed or is guaranteed to be notified in the future. When a data item is closed, the process receives a list of rule engines that must be notified regarding the closure of that item, which allows progress to continue.

To carry out the execution of a user script, components interact as shown in Figure 5. As discussed in §2, most processes act as workers, with about 0.01% of the processes acting *control processes*: servers or engines. First, the user script is evaluated by the first rule engine ①.

When a distributed loop construct is encountered, script fragments are created ② and posted to the task distribution system for load-balanced execution on other rule engines ③. Ultimately, calls to the user's external application programs of functions are produced and evaluated by workers ④. Data operations to perform the work may involve data services from multiple servers. On completion, output data items are closed ⑤ and rule engines containing dependent rules are notified ⑥.

6.3 Data Structure Representation and Management

Turbine script variable data is stored on servers and processed by engines and workers. These variables may be `string`, `integer`, `file`, or `container` data.

A variable of type `file` is associated with a string file name, however, unlike a `string`, it may be read before the file is closed. Thus, output file locations may be used in Turbine rules as *output* data, but the file name string may be obtained to enable the creation of a shell command line.

```
file a "input.txt"
file b "output.txt"
when { } then {b} from {create_file filename(a)}
when {a} then {b} \
     from {copy_file filename(a) filename(b)}
```

At the end of this given code, file `output.txt` is closed, allowing it to be used as the input to other Turbine rules.

Complex data structures may be constructed through the use of the core Turbine feature called *containers*. Container variables are similar to other Turbine script variables, but the value of a container variable is a mapping from keys to Turbine variable identifiers. Operations are available to insert, lookup, and list values in containers. This abstraction allows higher-level Swift language features such as structs, arrays, and stack frames to be represented in Turbine.

Turbine loops allow the user script to iterate over the keys and values in a container in a manner compatible with the Swift `foreach` construct. For simple loops over a range of values (conceptually, `do i = 1, 10`) a *range* container is constructed and is then used as the target of the iteration.

The stack in a Turbine program is represented as a container reference that is passed into a Turbine function. Thus, local function variables may be obtained by looking up the corresponding symbol in the container.

As an optimization, container keys are copied into the server responsible for the container. This reduces the number of remote data operations required for container operations used in iteration.

An additional optimization is the use of unevaluated expression strings to represent arithmetic operations. For example, consider the arithmetic case below:

Swift:

```
int a, b, c, d;
a = f(); b = g(); c = h(); d = i();
int x = (a+b)*(c+d);
trace(x);
```

Turbine:

```
integer a b c d
when { } then {a} from {f}
```

etc. ...

```
when {a b c d} then {x} \
     from {arithmetic "(_+_)*(_+_)" a b c d}
when {x} then { } from {print x}
```

A naive implementation of the given arithmetic expression would result in the creation of at least two temporary variables and two additional data dependency rules to represent the expression. However, the Turbine implementation can represent the whole expression with one rule, eliminating several messaging operations to the distributed data store and reducing the processing required.

7. Performance Results

Our performance results focus on three main aspects of Turbine performance: task distribution using ADLB, data operations using the new ADLB data services, and evaluation of the distributed Turbine loop construct. This provides a general picture of the ability of the meet the performance goals required by our applications.

All results were obtained on the SiCortex 5872. Each MPI process was assigned to a single core of a six-core SiCortex node, which runs at 633 MHz and contains 4 GB RAM. The SiCortex contains a proprietary interconnect with $\sim$1 microsecond latency.

The system has been successfully deployed on the IBM Blue Gene/P system and the Cray XT5 and XE6. Large-scale results from these systems are forthcoming.

7.1 Raw Task Distribution

To evaluate the ability of the Turbine architecture to meet its performance requirements, we first report the performance of a key underlying library, ADLB. Following that model, each ADLB server occupies one control process. We have measured the ability of ADLB to distribute tasks in a Turbine-like model from a single source. ADLB is configured with one server and a given number of workers. A single worker reads an input file containing a list of tasks for distribution over ADLB to other workers. This emulates a Turbine use case with a single rule engine that can produce rules as fast as lines can be read from an input file. Two cases are measured, one in which workers execute `sleep` for a zero-duration run (labeled "/bin/sleep"), and one in which they do nothing (labeled "no-op"). The results are shown in Figure 6. In the no-op case, for increasing numbers of client processes, performance improved until the number of clients was 384. At that point, the ADLB server was processing 21,099 tasks/sec, which far exceeds the desired rate of 1,000 tasks/sec per control process. The no-op performance is then limited by the single-node performance and does not increase as the number of clients is increased to 512. When the user task actually performs a potentially useful operation such as calling an external application (`/bin/sleep`), the single-node ADLB server performance in not reached by 512 client processes.

It should be noted that in practice, a Turbine application may post multiple additional "system" tasks through ADLB in addition to the "user" tasks actually required for the user application. Thus, the available extra processing on the ADLB server is appropriate. Overall, this indicates that ADLB is capable of supporting the system at the desired scale.

7.2 Data Operations

Next, we measure another key underlying service used by the Turbine architecture: the ADLB-based data store. This new component is intended to scale with the number of tasks running in the system; each task will need to read and write multiple small variables in the data store to enable the user script to make progress.

We have measured the performance of the ADLB data store and retrieve functionality added to the ADLB API for our system. A Turbine system is configured with a given number of servers and clients, and with one rule engine that is idle. ADLB store operations are performed on each client, each working on independent data. Each client interacts with different servers on each operation. Each client creates 200,000 small data items in a two-step process compatible with the dataflow model; they are first allocated and initialized, then set with a value and closed.

Figure 7 shows that for increasing numbers of servers and clients, the insertion rate increases monotonically. In the largest case, 1,024 servers were targeted by 1,023 workers with 1 idle rule engine, achieving an insertion rate of 19,883,495 items/sec. As data operations are independent, congestion only occurs when a server is targeted by multiple simultaneous operations, creating

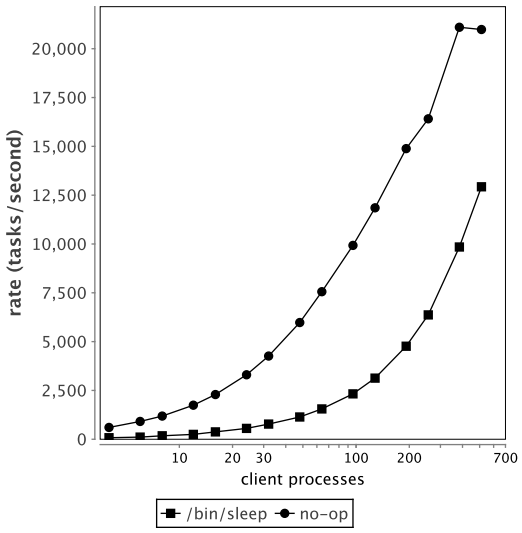

Figure 6. Task rate result for ADLB on SiCortex.

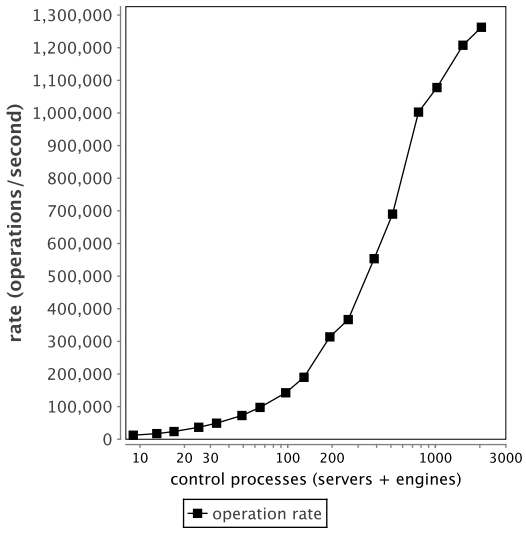

Figure 8. Range creation rate result for Turbine on SiCortex.

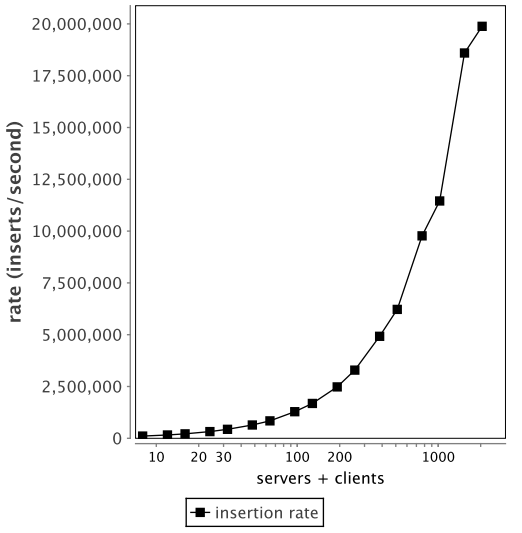

Figure 7. Data access rate result for ADLB on SiCortex.

a temporary hot spot. Continuing with the performance target of 1,000 tasks/sec per server, this allows each task to perform almost 20 data operations without posing a performance problem.

The data item identifiers were selected randomly, and thus, the target servers were selected randomly. Using the existing Turbine and ADLB APIs, an advanced application could be more selective about data locations, eliminating the impact of hot spots. This case does not measure the performance of data retrieval operations, but that is covered implicitly in §7.4.

7.3 Distributed Data Structure Creation

Thus far we have only measured the performance of underlying services. Here, we investigated the scalability of the distributed rule processing system, the engine processes. The engines are capable of splitting certain large operations to distribute rule processing work. An important use case is the construction of a distributed container, which is a key part of dataflow scripts operating on structured data. The underlying operations here are used to implement Swift's range operator, which is analogous to the colon syntax in Matlab, e.g., `[0:10]` produces the list of integers from 0 to 10. This can be performed in Turbine by cooperating rule engines, resulting in a distributed data structure useful for further processing.

We measured the performance of the creation of distributed containers on multiple rule engines. For each case plotted, a Turbine system was configured to use the given number of engines and servers. A Turbine distributed range operation was issued, creating a large container of containers. This triggered the creation of script variables storing all integers from 0 to $N \times 100,000$, where N is the number of Turbine rule engine processes. The operation was split so that all rule engines were able to create and fill small containers that were then linked into the top-level container. Workers were not involved in this operation.

Figure 8 shows the number of cooperating control processes, increasing to the maximal 2,048, half of which act as ADLB servers and half of which act as Turbine engines, does not reach a performance peak. Each operation results in the creation of an integer script variable for later use. At the maximum measured system size, the system created 1,262,639 usable script variables per second, in addition to performing data structure processing overhead, totaling 204,800,000 integer script variables in addition to container structures.

7.4 Distributed Iteration

Once the application script has created a distributed data structure, it is necessary to iterate over the structure and use it as a input for further processing. For example, once the distributed container is created in the previous test, it can be used as the target of a `foreach` iteration by multiple cooperating rule engines.

To measure the performance of the evaluation of distributed loops on multiple rule engines, for each case plotted, a Turbine system was configured to use the given number of engines and servers. In each case, a Turbine distributed range operation is issued, which creates a large container of containers. The operation is split so that all rule engines are able to create and fill small containers that are then linked into the top-level container. Then, the rule engines execute no-op rules that read each entry in the container once. The measurement was made over the whole run, capturing range creation and iteration. Thus, each "operation" measured by the test is

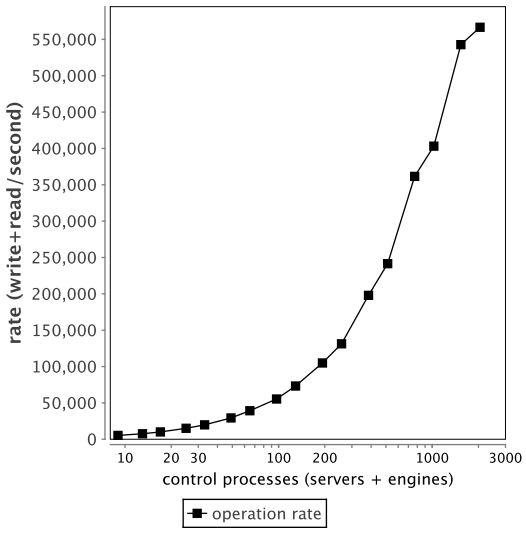

Figure 9. Range creation, iteration rate result for Turbine on SiCortex.

a complex process that represents the lifetime of a user script variable in a Turbine distributed data structure.

As in the previous test, each engine creates 100,000 variables and links them into the distributed container, over which a distributed iteration loop is carried out. Figure 9 shows that performance increases monotonically up to the largest measured system, containing 1,024 servers and 1,024 engines, in addition to 1 idle worker. At this scale, Turbine processes 566,685 operations per second.

This rate may be considered 277 operations/sec per control process, which approaches but falls below the performance requirement of 1000 operations/sec. per control process for our exascale projection. However, there are multiple ways that performance could be improved. First, this test was performed on the individually slow SiCortex processors. Additionally, we plan multiple optimizations to improve this performance. As noted in the previous test, variables are created in a multi-step manner compatible with our dataflow model. Since we are creating "literal" integers, these steps could be replaced with composite operations that allocate, initialize, and set values in one step. Multiple literals could be simultaneously set if batch operations were added to the API. The loop splitting algorithm itself is targeted for optimization and better load balancing. As a last resort, we could use more than 0.01% of the processes as control processes.

8. Status and Future Work

In this paper, we reported on the status of the Turbine intermediate code (TIC), which is a functional prototype, but is intended only for use by a code generator. We have prototyped a rudimentary Swift parser capable of emitting Turbine intermediate code, and will replace that with a fully functional compiler to create an implementation of Swift over Turbine. Some required features of the target languages (Skywriting, PyDflow, Dryad, and Swift) are not yet implemented. Swift's ability to start operations on the members of an array in a `foreach` loop as they are inserted before the array is closed is an important planned extension.

We plan multiple performance improvements to the underlying messaging used by Turbine, including the use of several composite operations to improve performance in commonly encountered cases. Data locality could be better exposed to the user to improve data access operations, and tasks could be bundled to reduce load on the control processes.

Additionally, we plan several usability features, including filesystem access operations and ease of use features to support integration with user native code functions. We would like to investigate fault tolerance in the model- dataflow program are naturally fault-tolerant, and in the case of total program shutdown, it is relatively easy to restart from any disk-resident data items (as in `make`). We intend to consider logging, checkpointing, and redundancy as possible paths to make Turbine programs fault-tolerant.

9. Conclusion

The contributions of this work are as follows.

First, we have identified many-task, dataflow programs as a highly useful model for many real-world applications, many of which are currently running in Swift. We provided projections of exascale parameters for these systems, resulting in requirements for a next-generation task generator.

Next, we identified the need for a distributed memory system for the management of the task generating script. We identified the *distributed future store* as a key component, and produced a high performance implementation. This involved the development of a dependency processing engine, a scalable data store, as well as supporting libraries to provide highly scalable data structure and loop processing.

Finally, we reported the performance results from running the core system features on the SiCortex. The results show that the system is within striking distance of the required performance for extreme cases.

While the TIC execution model is based on the semantics of Swift, it is actually much more primitive. We believe that it is capable of executing the data flow semantics of languages such as Dryad, Ciel, PyDflow, and Swift, and could thus serve as a prototype for a common intermediate code for these and similar languages.

Acknowledgments

Authors: Justin M. Wozniak, Ketan Maheshwari, Zhao Zhang, Todd Munson, Ian Foster, Dan Katz, Ewing Lusk, Matei Ripeanu, Michael Wilde

This research is supported in part by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research under contract DE-AC02-06CH11357, FWP-57810. Computing resources were provided by Argonne National Laboratory.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *IEEE Computer*, 19(8):26–34, 1986.

[2] T. G. Armstrong. Integrating task parallelism into the python programming language. Master's thesis, The University of Chicago, May 2011. URL `http://people.cs.uchicago.edu/~tga/pubs/armstrong-masters.pdf`.

[3] B. A. Allan, R. Armstrong et al. A component architecture for high-performance scientific computing. *Int. J. High Perform. Comput. Appl.*, 20:163–202, May 2006. ISSN 1094-3420. doi: 10.1177/1094342006064488.

[4] P. Beckman, I. Foster, M. Wilde, and I. Raicu. SWIFT: Scalable parallel scripting for scientific computing. *SciDAC Review*, (17), 2010.

[5] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. In *Proc. International Parallel and Distributed Processing Symposium*, 2011.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492.

[7] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. *Parallel Processing, International Conference on*, 0:586–593, 2008. ISSN 0190-3918. doi: 10.1109/ICPP.2008.44.

[8] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The pcn approach. *Sci. Program.*, 1:51–66, January 1992. ISSN 1058-9244.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007. ISSN 0163-5980. doi: 10.1145/1272998.1273005.

[10] Z. Li and M. Parashar. Comet: a scalable coordination space for decentralized distributed environments. In *Second International Workshop on Hot Topics in Peer-to-Peer Systems, HOT-P2P 2005*, pages 104–111, 2005.

[11] E. L. Lusk, S. C. Pieper, and R. M. Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, 17, January 2010.

[12] M. D. McCool. Structured parallel programming with deterministic patterns. In *Proc. HotPar*, 2010.

[13] D. G. Murray and S. Hand. Scripting the cloud with Skywriting. In *HotCloud '10: Proceedings of the Second USENIX Workshop on Hot Topics in Cloud Computing*, Boston, MA, USA, June 2010. USENIX. URL `http://www.usenix.org/event/hotcloud10/tech/full_papers/Murray.pdf`.

[14] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. NSDI*, 2011.

[15] Object Management Group. CORBA component model, 2002. `http://www.omg.org/technology/documents/formal/components.htm`.

[16] H. Pan, B. Hindman, and K. Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, page 6, Berkeley, CA, USA, 2009. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1855591.1855602`.

[17] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4): 277–298, 2005.

[18] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 22:1–22:12, Piscataway, NJ, USA, 2008. IEEE Press.

[19] SwiftR: a parallel and distributed computing package for R. SwiftR: a parallel and distributed computing package for R. `http://people.cs.uchicago.edu/~tga/swiftR/`.

[20] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, 2011.

[21] User info for SwiftR: a parallel and distributed computing package for R. User info for SwiftR: a parallel and distributed computing package for R. `http://www.ci.uchicago.edu/wiki/bin/view/SWFT/SwiftR`.

[22] V. Sarkar et al. ExaScale software study: Software challenges in extreme scale systems. DARPA Report, 2009.

[23] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13 (8-9), 2001.

[24] E. Walker, W. Xu, and V. Chandar. Composing and executing parallel data-flow graphs with shell pipes. In *Workshop on Workflows in Support of Large-Scale Science at SC'09*, 2009.

[25] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42(11), 2009.

[26] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37 (In Press, published online):633–652, 2011. doi: DOI: 10.1016/j.parco.2011.05.005.

[27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI)*, Dec 2008.

[28] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html`.